NOS-CPP - A Network Object System for C++

Benson Ma, Bo Han, Kenny Root

NOS-CPP enables software developers to easily make their C++ project distributed among multiple systems. The primary design goal is simplicity of implementation. Retrofitting NOS-CPP to an existing C++ codebase need only change class instantiation sites--NOS-CPP handles façade class generation, garbage collection, lease renewal, serialization, and network communication.

Interface and Design

Design Goals

NOS-CPP is based on Network Objects by Birrell et al [1]. NOS-CPP follows these principles:

1. Simplification of the stub code generation process
2. Simplification of the type system
3. Simplification of the underlying RPC protocol (i.e. the RPC schema)

Terminology

NetObj refers to a user-defined network object class.
AgentObj refers to a NOS-CPP-generated façade class used by the *NOSAgent*.
ServerObj refers to a NOS-CPP-generated façade class used by *server code*.
ClientObj refers to a NOS-CPP-generated façade class used by *client code*.
Server code refers to user code that exports *ServerObj* instances.
Client code refers to user code that imports *ClientObj* instances.
The *NOSAgent* manages *AgentObj* instances created from *ServerObj* instances exported by *server code*. The *NOSAgent* handles remote method invocations and garbage collection.
The *NOSClient* manages *ClientObj* instances imported by *client code*. The *NOSClient* renews leases on the *ClientObj* instances it still references. *ClientObj* method invocations call *NOSClient* network communication methods.

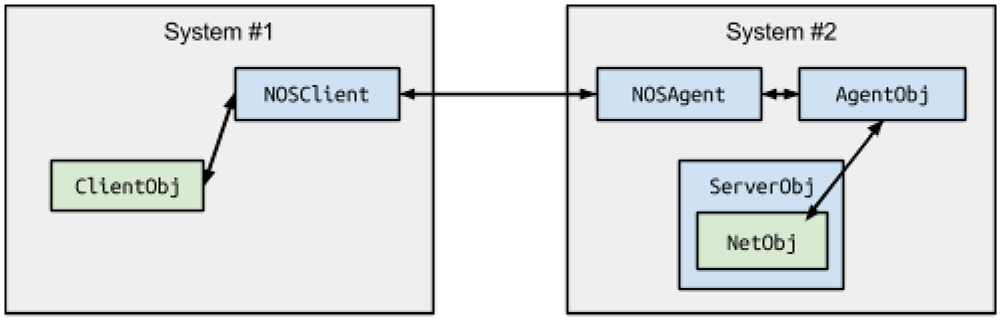

Figure 1: Overview of a dispatch from *client code* to *server code*

Interface description language (IDL)

Instead of a traditional IDL language, NOS-CPP utilities don't require anything special beyond the C++ class definition files to generate the necessary code. The code generator scans classes extending the NetObj class using the Clang[5] API. The code generator inserts the information about all public methods and fields into a template defined using the CTemplate[6] language, which generates the façade and ancillary classes to make a *NetObj* instance appear local to the software developer.

Code generation
Given every *NetObj* class, NOS-CPP's code generator generates facade *ServerObj, AgentObj,* and *ClientObj* classes. The purpose of each of these generated classes is as follows:
ServerObj inherits *NetObj* and is exportable by *server code*. *ServerObj* wraps a *NetObj* instance, which *ServerObj* instantiates. *ServerObj* implements all *NetObj* methods by returning the result of calling the method of the stored *NetObj* instance with the same arguments.
AgentObj is used by the *NOSAgent* to track exported variables. Exporting a *ServerObj* results in the creation of an *AgentObj* using the *Agent Registrar* (described below).
ClientObj is importable by *client code*. *ClientObj* contains the variable name and information to communicate with the *NOSAgent* storing the variable. The *client code* with a *ClientObj* can pretend that it has a *NetObj*--with a caveat on getting/setting public fields (described below).
For all *NetObj* classes, NOS-CPP generates a single *Agent Registrar* class and a single *Client Registrar* class. Both *Registrar* classes define a *constructor method* wrapping the constructor for each *AgentObj* and *ClientObj* type. The *Registrar*s', at initialization, map runtime type information (RTTI) names to these *constructor methods*. This enables the *Registrar* to, given an RTTI name for a specific *ServerObj* or *AgentObj*, respectively return a corresponding *AgentObj* or *ClientObj*.
Implementation
RPC Schema
Client code talks to *server code* via **`RPCRequest`** messages and receive responses to requests via **`RPCResponse`** messages. Serialization methods are built into the **`RPCRequest`** and **`RPCResponse`** classes using the **`cereal`** library so that they can be transported over TCP and constructed on-the-fly from incoming application-level TCP packets. We have been able to design a simplified and consistent message schema for the RPC protocol using the **`Serializer`** and **`TupleFunctional`** libraries discussed below. Their schemas are as follows:
```
// RPCRequest
{
	Type:     enum { IMPORT, INVOKE, RENEW_LEASE, },
	ClassID:  uint32_t,
	ObjectID: std::string,
	MethodID: uint32_t,
	Body:     std::string,
}
// RPCResponse
{
	ServerCode: enum { OK, OBJECT_NOT_FOUND, FAIL, },
	Body:       std::string,
}
```
The **`Body`** field in an **`RPCRequest`** is simply the serialized `std::tuple` of arguments to the object's method to be called on the server, while the **`Body`** field in an **`RPCResponse`** is the serialized return value of a remote method invocation, or an exception string if the server throws a **`ServerCode`** of **`FAIL`**.
NetObj construction
An exportable *NetObj* is created in *server code* by constructing a *ServerObj* with the same arguments as those defined in the *NetObj*. For example, if *NetObj* class Foo's constructor was `Foo(int32_t num)`, the *server code* calls `FooServer(5)` to get a FooServer containing a `Foo(5)`.

Export
Server code calls `NOSAgent::Instance()->Export("foo", obj)`, where `"foo"` is the name of the exported object, and `obj` is a *ServerObj*. An *AgentObj* containing `obj`'s *NetObj* instance is created using the *Agent Registrar*: `obj`'s RTTI name is used to invoke the appropriate *constructor method*. The *NOSAgent* contains a map of exported variable names to *AgentObj* instances.
Import
ClientObj instances are created using `NOSClient::Instance()->Import<FooClient>("foo", address, port)`, where `"foo"` is the variable's name, and `address` and `port` identify the *NOSAgent*. `Import` makes a `get_type` request to the *NOSAgent* asking for `"foo"`'s RTTI name. The *NOSClient* uses the *Client Registrar* to invoke the appropriate constructor method. The *ClientObj* stores the name of the imported variable and the address and port of the *NOSAgent*.
Garbage collection
Lease-based garbage collection is used. The garbage collector (GC) runs on a thread created by *server code* on *NOSAgent* initialization. The GC scans the *NOSAgent*'s map of exported variable names to *AgentObj* instances and garbage collection metadata. Garbage collection metadata is specific to each *AgentObj* and contains a boolean indicating whether the corresponding *ServerObj* has been destroyed and a timestamp indicating the last time that time *client code* has renewed the lease on this *AgentObj*. If the boolean is true and the timestamp is more than a user-configured *lease period* behind the current time, the mapping for the name is unset and the *AgentObj* is destroyed, destroying also its stored *NetObj*.
When a *ServerObj* is destroyed, the destructor notifies the *NOSAgent* of its own destruction, thereby setting the garbage collection metadata boolean to true. The *ServerObj* does not destroy its *NetObj*--that task is left to the corresponding *AgentObj*'s destructor.
Every `x` seconds, where `x` is less than the user-configured *lease period*, a *NOSClient* thread makes `renew_lease` requests to the *NOSAgent*(s) storing its imported variables.
For the GC to prematurely reclaim an *AgentObj*, the *server code* must have deleted its *ServerObj* and all *client code* that refer to the variable must be partitioned from the *NOSAgent* for longer than the *lease period*. To tune the GC's aggressiveness, the *lease period* can be shortened or lengthened.
Network Layer
NOS-CPP's RPC protocol is implemented on top of TCP. While the NOS-CPP network layer was built using the **tcpsockets** library[2], an application-level packet-oriented protocol was implemented above TCP to support RPC, as TCP is a stream-oriented protocol. In this protocol, the sender first sends a *packet header*--a byte-stream of fixed size (determined at compile-time) that contains a NOS-CPP protocol signature string and the byte-representation of the actual message packet's byte-size. On accepting a TCP connection, the receiver reads in the header byte-stream to verify the NOS-CPP protocol signature and determine the size of the message it will receive. The receiver allocates the buffer appropriately and copies the stream contents up to the pre-determined buffer size using **read()**. Timeouts for each TCP connection differentiate a failed node and a slow network. A simplified implementation is given in the Appendix.
Data Serialization and Marshalling
The **`Serializer`** module built on top of the **`cereal`**[4] C++11 header-only library performs data serialization. **`cereal`** serializes all POD datatypes and STL containers, while providing mechanisms for serializing and deserializing user-defined types and classes. **`cereal`** is portable across C++11

compilers, and its binary serialization format is portable across machines, accounting for machine endianness. The **Serializer** module provides the following API:

```
namespace Serializer {
    // Packs an object into a byte-array wrapped by a std::string
    template<typename T> std::string pack(const T& obj);
    // Unpacks an object from a byte-array wrapped by a std::string
    template<typename T> T unpack(const std::string &serialized);
};
```

Method Invocation/Function Application

S*erver code* invokes *NetObj* methods through the corresponding *ServerObj* facade class, which directly calls the method on the *NetObj* instance.

Client code invokes *NetObj* methods though the *ClientObj* class, which sends the variable name, the arguments marshalled using the **Serializer** module, and a code identifying the method to the *NOSAgent*. The *NOSAgent* unmarshalls the arguments and invokes the method on its *AgentObj*'s *NetObj* instance using the process described below. The *NOSAgent* marshalls the return value into the response, which *ClientObj* method then unmarshalls to the caller.

To reduce code complexity of the IDL compiler, a different programming methodology is required to simplify the generated server-side code such that it will look as if it were written in a dynamic programming language. In general, server-side processing of network object RPC requests contain the following steps:

1. Based on the RPC request contents, look up the object
2. Based on the RPC request contents, look up the method call
3. Based on the object and method call, de-serialize the method arguments properly
4. Invoke the method call on the object, and pass the arguments into the method.

Steps 3 and 4 are heavily dependent on the method parameter and return type signature, which make it impossible for the IDL compiler to generate efficient and generic code, since it requires generating custom code for de-serialization, argument extraction into temporary variables, and correct invocation of the method with the temporary variables. To address this, it is desirable to be able to apply the packed arguments contained in an RPC request *directly* onto a function call, and leave the code for unpacking arguments for the compiler:

```
double func(/* arguments to a function call */) { /* implementation */ }
auto arguments = std::make_tuple(/* values for a function call */);
double x = apply_fn(func, arguments); // Apply arguments directly without unpacking
```

The **apply_fn** function is similar to the **apply()** functional programming construct found in languages such as Lisp or Python. This is implemented using C++11 template meta-programming techniques, and we built the **TupleFunctional** library to support this feature. The library has the following API:

```
namespace TupleFunctional {
    // Apply a tuple's contents as arguments to a static function
    template<typename Function, typename Tuple>
    auto apply_fn(Function&& f, Tuple&& t);

    // Apply a tuple's contents as arguments to a non-static member function
    template < typename FN, typename TYPE, typename... ARGS >
    auto apply_nonstatic_fn(FN&& fn, TYP&& obj, std::tuple<ARGS...> &args);
};
```

The compiler recursively unpacks the tuple and applies the values of the arguments tuple directly to the function call, as well as type-checks the values during compile-time. No data copying is done, since the implementations of **`apply_fn()`** and **`apply_nonstatic_fn()`** employ C++11 **`move`** semantics. This library immensely simplifies the compiler's code-generation logic, as the compiler does not need to manage the RPC message extraction details that are customized for each object method signature. As an example, the compiler-generated server code that handles requests for remote **`Foo`** objects contains code that looks like the following:

```
  /* Switch statement */
  case FooMethodID::bar_method: {
    // De-serialize method arguments to std::tuple
    auto args = Serializer::unpack<std::tuple<int,double>>(request.Body);
    // Apply args tuple to method invocation
    auto result = TupleFunctional::apply_nonstatic_fn(&Foo::bar_method, object, args);
    // Pack the results into the RPC response message
    response.Body = Serializer::pack<decltype(result)>(result);
    response.Code = ServerCode::OK;
    break;
  }
```

In fact, the only code difference in IDL compiler-generated output between this method stub example and another method's stub is the type parameters around the API call for unpacking the `arguments` byte-array into a `std::tuple`. As a result, the template file employed by the compiler for generating code is also extremely short and manageable.

Public Fields

The IDL compiler automatically generates **`getter`** and **`setter`** methods for each public attribute defined in the *NetObj*. The serialization and generic function application APIs have made the code extremely straightforward to generate These methods follow C++ **`copy`** semantics to avoid under-the-feet deletions/invalidations of user-created objects and to make the generated code uniform across POD and non-POD datatypes. However, our code-generation engine and templating system is sufficiently simple enough that we can set the compiler to easily generate for each non-POD-type public attribute an extra set of **`getter`** and **`setter`** methods that follow C++ **`move`** semantics.

Evaluation

NOS-CPP has been used to build a simple distributed key-value store in few lines of user code, and other user programs such as a chat program are in development. More details are forthcoming.

Example Application - Remote Key Value Store

We were able to build a remote key-value store server in less than an hour using the NOS-CPP framework. In fact, most of the development time was spent writing the client side command-line interpreter interface for the key-value store to showcase the application; the actual programming of the key-value store NetObj class took less than one-third of the total development time and the library generation was instantaneous. This goes to show how NOS-CPP can be used to rapidly prototype and deploy distributed-network-object-based services.

Future work

Locate

Rather than providing the IP address and port of the servers they would like to talk to, clients should be able to find the server that provides the requested remote objects during import. Future implementations of NOS-CPP can include a `Locate` function that maps a name to an IP address and port. Servers exporting NetObj objects would register with some nameserver, and clients would query the nameserver prior to importing. The use of a nameserver alongside server replication will provide better fault-tolerance for dealing with NetObj server crashes.

More argument and return types

While fields in NOS-CPP objects are currently restricted to POD types, `std::string`, and STL containers over the simple types, it is not difficult to relax this restriction and allow more complex user-defined datatypes as fields when declaring a new network object class. In fact, the only requirement to enable this is that users who have complex fields in their network object definitions must define serialization methods for the associated classes. This can easily be done using the **`cereal`** API. In addition, non-POD types with pointer fields can also be serialized using the **`cereal`** API, provided that the pointers are wrapped in a `std::shared_ptr` and objects pointed to also have serialization methods defined.

Conclusion

The NOS-CPP framework provides the tools for software developers to easily build distributed software as well as make existing C++ projects distributed among multiple systems easily. While providing similar functionalities as many many network-object systems, NOS-CPP differs from the rest in that it was designed from the start to be simple to use for the end-users. Through the use of a simple RPC schema, a unified remote method invocation mechanism based on C++11 metaprogramming techniques, and the equivalence of C++ header files with the IDL using the Clang API, we have been able achieve this, whereby a software developer has essentially no learning curve to overcome in order to build a distributed software project. The implementation currently supports Linux and Mac OS X, and the source code is available at https://github.com/bhan/nos-cpp/.

Acknowledgements

We thank the various peers we have come across on StackOverflow, Google Code, and GitHub who have indirectly provided advice and example code for ideas that we have experimented with and/or adopted into the design of NOS-CPP.

References

1. Andrew Birrell, Greg Nelson, Susan Owicki, Edward Wobber, Network Objects, Research Report 115, Systems Research Center, Digital Equipment Corporation, Palo Alto, February, 1994.
2. Tcpsockets. https://github.com/vichargrave/tcpsockets, November 2014.
3. Pinty, toy prototype for a distributed computing system in C++. https://github.com/darabos/pinty, November 2014.
4. Cereal - A C++11 library for serialization. http://uscilab.github.io/cereal/, November 2014.
5. clang: a C language frontend for LLVM. http://clang.llvm.org, November 2014.
6. CTemplate: powerful but simple template language for C++. https://ctemplate.googlecode.com/, November 2014.

Appendix

Network layer simplified implementation

```
uint32_t send_packet(TCPStream *stream, const std::string &buffer) {
	// Send header info (size of the packet coming down the pipe)
	send_header(stream, buffer);

	// Send the actual data (in chunks of fixed size)
	uint32_t num_packets_sent = send_packet_in_fixed_sized_chunks

	return num_packets_sent;
}

void receive_packet(TCPStream *stream, std::string &dest_buffer) {
	// Read the header to determine the size of the incoming application packet
	uint32_t expected_buffer_len = receive_header(stream);

	// Resize buffer to the expected application packet length
	dest_buffer.resize(expected_buffer_len);

	// Copy from TCP stream in fixed-sized chunks over to the buffer
	copy_from_stream_up_to_expected_size(dest_buffer, expected_buffer_len);
}
```

Abandoned Initial RPC Designs for NOS-CPP

Our initial design for NOS-CPP involved a system architecture that was designed to be much more aggressive in making the RPC protocol as generic as possible and significantly reducing the complexity of RPC code generation from the IDL compiler. We envisioned an RPC protocol in which *serializable closures* were passed as messages between the client and server, where the stated computation is executed on the recipient to generate the desired state machine changes or output the return value of an operation. Under this design, the client creates a closure that takes as arguments only the pointer to the remote surrogate object on the server. When invoked on the server, the closure will generate an **`RPCResponse`**, which contains a serialized closure that when invoked, will return the output of the object's method call. This significantly reduces the complexity of the RPC schema, since all RPC message bodies are simply serialized `std::function` objects. A full code example is provided below of what a client-side method stub would look like under this architecture:

```
uint32_t FooClientProxy::bar_method(std::vector<std::string> &val1, bool &val2) {
	// Create a function that generates the closure
	auto delegate_method_gen = []( std::vector<std::string> &closure_val1,
	                               bool &closure_val2 )
	                               -> Closure<RPCResponse(FooServerProxy*)>
	{
	  // Capture the variables by value, NOT reference
	  return [=](FooServerProxy *obj){ return obj->bar_method(
	                                        closure_val1, closure_val2); };
	};
```

```
        // Generate the closure
        auto delegate_method = delegate_method_gen(val1, val2)

        auto request = serialize( {
            ClassID:  this._classID,
            ObjectID: this._objectID,
            Action:   delegate_method.Serialize(),
        } );

        RPCResponse response = rpc_send(request);
        if (response.server_code != OK) {
            throw new Exception(response.server_code);
        }

        // De-serialize the callback closure and execute,
        // which will output the desired uint32_t
        return Function<uint32_t()>::Load(request.callback_method)( );
}
```

While this design can be very powerful, we abandoned it for two reasons. The first is that this protocol entails very deep security implications, since malicious clients can send closures with arbitrary malicious code over to the server to induce service-compromising attacks. Furthermore, there is no code-verification system possible that we can implement on NOS-CPP to perfectly prevent malicious code execution, since this amounts to solving the Halting Problem. Additional security protocols can be implemented to limit the execution rights of a closure, but this will bring upon additional complexity to NOS-CPP.

The second problem, which is the immediate reason behind our abandonment of the idea, is that this design is not implementable in C++. We initially experimented with this idea using an external C++11 template library that casts `std::function` types into "serializable" template classes [3]. What we have found is that while the closures can execute perfectly with the captured variables after serialization and de-serialization, this feature is limited to use within a single process space; the program that creates the serialized closure and the program that executes the closure must be the exact same compiled binary. Furthermore, this feature will only work with Address Space Layout Randomization (ASLR) disabled, either through an environment flag or compilation flag depending on the operating system. These two limitations are known to the developer of the library, whom we have communicated with. We have attempted to circumvent the exact-binary limitation by placing the closure creation and execution code into a common software library before linking it against the agent and client binaries, but this method did not work. This comes as a surprise to us, since `std::function`s are objects of generated classes instantiated by the compiler during compilation. It is unclear whether this defect arises because the C++ language does not have a defined ABI, or because the C++ Standard does not mandate `std::function` and lambda types to be well-defined enough for serializability.